CS 315 Data Structures **Fall 2012**

Lab # 3 Image synthesis with EasyBMP

Due: Sept 18, 2012 (by 23:55 PM)

EasyBMP is a simple c++ package created with the following goals: easy inclusion in C++ projects, ease of use, no dependence on other libraries (totally self-contained), and cross-platform compatibility. The goal of this lab project is to learn about the functions in EasyBMP package and work on two image creation problems. The first one is to create a checker-board pattern shown in Figure 1 below.

Figure 1

The second one is to implement a recursive algorithm that takes as input an image I, and creates another image J of the same size as I. The relationship between J and I is best explained by the example shown below. The image on the left-side is the input image and the one on the right is the output J.

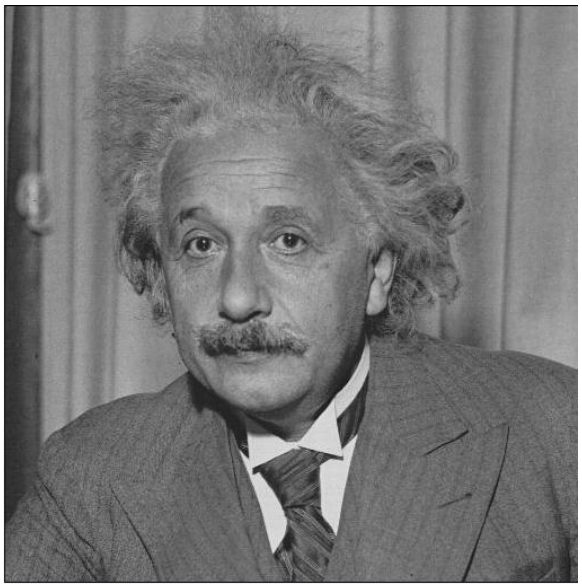


Figure 2(a): Input image

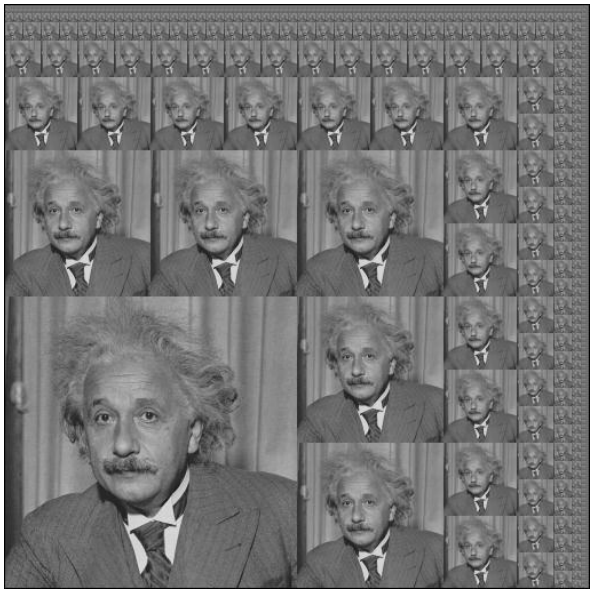


Figure 2(b): Output image

You can assume that the input image is square-shaped, and its size is a power of 2.

Goals of the project:

- learn to use a C++ library designed for image manipulation. (We will use this library in other projects as well.)
- learn to synthesize a simple image by setting the color components of each pixel.
- learn to use recursion to generate a tiling of an image that scales down by a factor of 2 along each row and each column.

EasyBMP library:

C++ does not directly support image objects/data types in its language specification. However, several external libraries have been built for representing and manipulating images using c++. EasyBMP is one such library written in C++ that provides support for opening an image (in the BMP, or bitmap format), and manipulate the individual pixels of the image. Although there are more powerful packages, the reason for choosing this package is two fold: (a) it is very easy to learn and (b) the code size of the library is very small. In addition, it is cross-platform compatible. Later when we discuss some applications that require a more powerful library, we may consider **CImg**, **OpenCV** or other packages.

You don't have to know the details of how a BMP image is stored in a file. It is enough to know how to make calls to EasyBMP functions. The following are some useful classes and functions from the EasyBMP library:

- *RGBApixel*
  A class that represents a single colored pixel. Each pixel is determined by four color components -- red, green, blue, and alpha. Each of these color components is an unsigned char variable (i.e. an integer between 0 and 255), and is accessed as follows: if v is a pointer to an RGBApixel variable, then the relevant color components can be accessed using v->Red, v->Green, v->Blue, and v->Alpha, respectively.
- *BMP*
  A class that holds a bitmap image, i.e. a rectangular array of colored pixels (i.e. RGBApixel objects). Once you define a variable of type BMP, you can then manipulate it using the functions listed below.
- bool *BMP::ReadFromFile(const char* FileName)*
  Given the name of an image file, this function loads the image into the BMP class, so that your program can then access the image using member functions listed below. The function returns a bool value indicating whether the read was successful.

- bool *BMP::WriteToFile(const char* FileName)*
  If you want to save the image in your program to disk, this function allows you to do so; you specify the desired filename as the argument to the function. The function returns a bool value indicating whether the write was successful.
- int *BMP::TellWidth()*
  This member function takes no inputs, and returns the number of pixels in the horizontal direction.
- int *BMP::TellHeight()*
  This member function takes no inputs, and returns the number of pixels in the vertical direction.
- void SetSize(int, int) This sets the width and height of an image to the two specified parameters.
- RGBApixel* *BMP::operator()(int i, int j)*
  This is an example of an overloaded operator. Here the parentheses have been overloaded, so that given img is an object of type BMP, the expression img(i,j) returns a pointer to the color information stored at the pixel whose *x*-coordinate is i and whose *y*-coordinate is j. The upper-left-hand pixel is given by *i* = 0, *j* = 0 and the positive x-axis is West to East while the positive y-axis is from North to South.

Another useful function is *RangedPixeltoPixelCopy*. This will be explained in the lab.

Here are the first steps to learn how to use EasyBMP. First download the EasyBMP package using the link http://easybmp.sourceforge.net/download.html. Next download the user manual from link http://easybmp.sourceforge.net/documentation.html. The manual presents some examples.

In the following, a simple example is presented to illustrate the use of *EasyBMP*.

Suppose we want to convert a color image or a gray scale image into black and white. This is a tricky process, but a very important one – for example when we want to print a color photo on a black and white printer, we need such an algorithm. We will use a very simple-minded algorithm called *thresholding*. Recall that each color pixel is defined by three color components (R, G, B), each taking a value between 0 and 255. R = G = B = 255 is white, R = G = B = 0 is black. In thresholding, the average the color components of a pixel (i.e., compute (R + G + B)/3) is computed and if this average exceeds 127, this pixel to mapped to white, otherwise it is mapped to black. In the code below, we take a weighted average by weighting the colors by 0.3, 0.6 and 0.1 (since the human eye's sensitivity to Green is more pronounced than that for Red, and is least sensitive to Blue.) We present the code to implement this algorithm below.

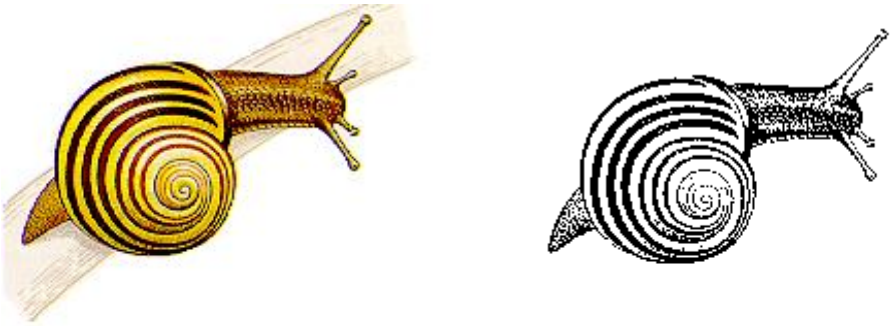
```
#include "EasyBMP.h"
using namespace std;

void colorToGray(BMP & Output) {
  int picWidth = Output.TellWidth();
  int picHeight = Output.TellHeight();
```

```
  Output.SetBitDepth(1);  //compression happens here
  for (int i = 0; i < picWidth-1; ++i)
     for (int j = 0; j < picHeight-1; ++j) {
           int col = 0.1* Output(i, j)->Blue +
             0.6*Output(i,j)->Green +0.3* Output(i,j)->Red;
            if (col > 127) {
             Output(i,j)->Red = 255;
               Output(i,j)->Blue = 255;
             Output(i,j)->Green = 255;
            }
          }
   }

int main( int argc, char* argv[])
{
 BMP myImage;
 myImage.ReadFromFile(argv[1]);
 colorToGray(myImage);
 myImage.WriteToFile(argv[2]);
 return 0;
}
```

A sample input/output is shown below.



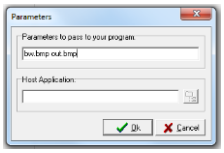
The resulting b/w image on the right is much smaller in size. In addition to its use in reducing the file size, this program can also used to create a pencil-sketch effect of a photograph.

To run this program, please follow the steps outlined below:

1) create a file bw.cpp containing the above code.
2) place the files `EasyBMP.h, EasyBMP_DataStructures.h, EasyBMP_BMP.h, EasyBMP_VariousBMPutilities.h and EasyBMP.cpp` in the

same directory. (Equivalently, using IDE such as Dev C++, add all the 5 files listed above and bw.cpp to a console application project.)

3) compile bw.cpp and create an executable bwout.
4) download an input image test.bmp into the same directory.
5) Run bwout using the format: .a/bwout test.bmp out.bmp (Equivalently, in Dev c++, select parameters option under "Execute" and in the dialog box displayed, enter the file names as shown in the figure:



Now choose "run" option under Execute.)

6) Now list the files in the directory and make sure that an image file named out.bmp has been created.
7) Open out.bmp using an image viewer and check that out.bmp is a black and white version of test.bmp.

Note that test.bmp can be a color image, although the in the above example, the image on the left is a gray-scale image.

Problems to be solved are formally stated below:

Exercise 1. Write a program that generates a 256 by 256 image of a checker-board containing 8 rows and 8 columns with alternating 32 by 32 square images of red and black colors.

Exercise 2: Generate a recursive tiling of an image I as in Figure 2a and generate an image as shown in Figure 2b.

Solution to Exercise 2:

It is clear that your program should create many copies of the input image, scaled to ½, ¼, 1/8 etc. sizes. Then you have to tile an image the same size as the input using copies of these images. The precise task **tile** is described below formally in the following steps:

Let I be the input image and O the required output image.

Let tile(I) denote the output image O given I as input. Also let Q(O) denote a quadrant of O. (Q may be NE = North East, SW etc.) Let the output when you apply *scale_down* to input I be S. Here is how each quadrant of O will be filled:

(1) The South-West (SW) quadrant of O is a scaled down (by a factor of 2) copy of I. Thus, this can be computed by a call to function scale_down with I as input.

(2) The NE(O) = NE quadrant of O is a scaled copy of O itself. Thus, NE(O) can be obtained by ***recursively*** calling tile(S).

So far, we have filled the SW and NE quadrants of the output O. Let the left half of NE(O) be I1.

(3) It can be seen that NW(O) is just two identical copies of I1 so just copy I1 twice to fill NW(O).
(4) Finally, note that SE(O) contains two identical copies of the lower half of NE(O). Thus, copying I2 twice into the correct place, SE(O) gets filled.

The recursion continues until a 1 x 1 image is reached which is the base case. In this case, we set O = I. (You can assume that the input image is a square image and its height and width are powers of 2 so you can apply the transformations described above without distortions.) Copying a part of O into another part of O (which is needed in steps 3 and 4) can be accomplished using the EasyBMP library function *RangedPixelToPixelCopy*.

Scaling down an image: To scale down an image of size 2k x 2k to an image of size k x k, you can implement the following algorithm: for all k, 0<= j <= k–1 (where k = the number of row pixels = the number of column pixels), the average of the pixels [2j,2j], [2j, 2j+1], [2j+1, 2j] and [2j+1, 2j+1] will be stored in pixel [j,j].

In all, you are responsible for writing just two functions:

- **`scale_down(BMP& I, BMP& O)`**
- **`tile(BMP& I, BMP& O)`**

The size of the complete program will be about 30 lines.

What should be submitted?

Include the source code, compiled executable code, along with two input images (an input image and the tiled output) and all the EasyBMP files – included in a single zip file. The output images should be labeled ***out1.bmp*** and ***out2.bmp***

Your program should check that the input image is of square shape and that its height (and width) is a power of two. It should print a message stating the input format is wrong and not call the tiling function in case of format error.

This lab will be weighted ***3 points***.

Grading:

Exercise 1 - correct output → **25 points**

Exercise 2 – error checking → **15 points** correct output → **60 points**